ERTOS: Enclaves in Real-Time Operating Systems

Alex Thomas
University of California Berkeley
alexthomas@berkeley.edu

Stephan Kaminsky
University of California Berkeley
skaminsky115@berkeley.edu

Dayeol Lee
University of California Berkeley
dayeol@berkeley.edu

Dawn Song
University of California Berkeley
dawnsong@cs.berkeley.edu

Krste Asanovic
University of California Berkeley
krste@berkeley.edu

Abstract

With the growing popularity of edge computing and Internet of Things (IoT) devices, there is an increased need for secure computation on embedded devices. Typically, embedded devices have a heterogeneous environment and do not have general security protections compared to hosts on the cloud. As we see more third-party libraries and applications being run on embedded devices, we face the risk of system compromise that even the device's RTOS kernel cannot protect. There is a need for creating Trusted Execution Environments (TEEs) on embedded devices; however, many current TEEs have expensive hardware requirements. We propose using Keystone, a framework for creating customizable TEEs, on RISC-V architectures. The hardware requirement for creating TEEs in Keystone are generally available on standard RISC-V devices as RISC-V already provides PMP registers, the basis of Keystone's isolation. We propose using Keystone with FreeRTOS to implement a module in FreeRTOS for creating efficient and dynamic TEEs on embedded devices. We introduce ERTOS, a new module to FreeRTOS that allows the creation of secure tasks that can be attested and strongly isolated from other tasks using Keystone's security monitor. ERTOS exposes an easy-to-use API that allows developers to create and run enclave-protected tasks. ERTOS adds negligible performance overhead for computation-intensive tasks inside an enclave and introduces optimizations to allow inter-task communication to be more efficient.

1 Introduction

With the rise of Internet of Things (IoT) devices, there is an increased amount of data being collected, from temperature information to audio input. Because of the ubiquity of IoT devices, a user will be required to trust more and more devices [16]. Moreover, some of these devices are critical to the safety of human lives like automobile or medical devices.

Unfortunately, many IoT devices do not put considerable effort into security and have a wide range of vulnerabilities [11]. We are starting to see attacks targeting embedded devices today. By using a flying drone, a group of researchers were able to compromise a Tesla vehicle using a privileged escalation attack on ConnMan, a commonly used embedded application to manage internet connections [5].

With real-time embedded devices, we are seeing the increased use of third-party applications like crypto libraries, intrusion-detection software, and more [22]. Some smart home devices have adopted third-party application support. To expand in more detail, let us focus on Samsung SmartThings devices, which typically run on a light-weight RTOS [6]. Users of Samsung SmartThings devices [3] have created communities to share custom automation rules that control how their device behaves [4]. For example, a user may share a potentially helpful automation rule that sends an alert if the temperature of one's household is outside a specified range. Users exchange GitHub repository links to install the third-party apps on their SmartThings instances with no application vetting process by Samsung.

With more platforms supporting third-party applications, there is increased need to isolate these applications. A general purpose OS would typically use a hardware memory management unit to provide memory isolation, but in the more resource-constrained environment of embedded devices, the memory isolation is ignored for more efficient performance.

We cannot rely on the security of the RTOS kernel as previous work found protection measures in place for the FreeRTOS kernel deficient [17]. Moreover, previous work has found FreeRTOS vulnerable to privilege escalation attacks [23]. This means solutions that use a Memory Protection Unit, which is configurable by the RTOS kernel, are rendered useless if the attacker is running on a privileged mode. Moreover, an application must inherently trust the large code base of the RTOS kernel. A large privileged code base increases the possibility of finding a vulnerability in the code. A solution to protect an embedded device application should also isolate and protect from an adversary RTOS kernel.

In this paper, we propose using Trusted Execution Environments on embedded devices. This fits our need for strong isolation and secure computation guarantees. Furthermore, TEEs provide mechanisms to attest an application to verify that the application running inside a TEE is not modified and is running on trusted hardware. We contribute a module to FreeRTOS that allows the dynamic creation of TEEs using Keystone with negligible computation overhead. Our module exposes a familiar API for enclave creation to developers that worked with FreeRTOS. We evaluate ERTOS based on

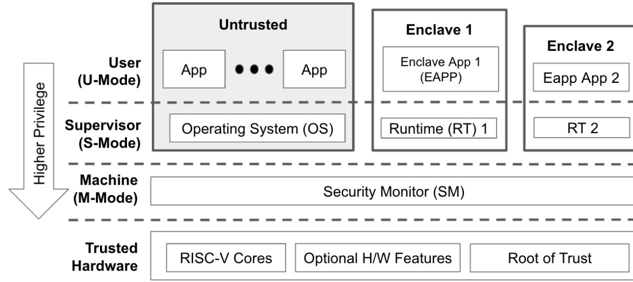

Figure 1. Visualization of Keystone's Software and Hardware Stack [14]

compute-bound benchmarks and inter-task communication. We also open the doors to a hardware agnostic approach for TEEs in embedded devices as our FreeRTOS module may be developed to possibly support any enclave back-end.

2 Background

The challenge of TEEs on embedded devices is that some hardware-enforced TEE solutions like SGX rely heavily on virtual memory support or other expensive hardware [10]. Furthermore, we need multiple isolated zones, unlike ARM TrustZone, which only offers a single isolated zone [19]. Multizone, a RISC-V based framework for creating TEEs, only supports statically created enclaves [18]. In order to prototype our system, we needed an enclave back-end that is compatible with embedded devices and allowed us to dynamically create enclaves.

2.1 Keystone

Keystone is an open-source framework for creating multiple, customized TEEs based on RISC-V architecture [14]. Keystone uses a privileged *Security Monitor* (SM), which is responsible for creating, deleting, and switching into enclaves dynamically. In order to create TEEs, Keystone utilizes *Physical Memory Protection* (PMP) registers, which act as base and bound registers that seal off memory from other entities including the privileged host OS. These PMP registers are generally available on standard RISC-V machines. Keystone guarantees the confidentiality and integrity of memory within the enclave. We provide a visualization of the different components involved in Keystone in Figure 1. Base Keystone does not protect against physical adversaries as pages aren't encrypted and integrity-checked like SGX [10]; however, Keystone does offer software-based page encryption and integrity checks as a module which requires no additional specialized hardware [7].

2.1.1 RISC-V Privileged ISA.

RISC-V provides several privilege levels to provide protection across several layers of the software stack. Keystone relies on **Machine**, **Supervisor**, and **User** mode [21].

Machine (M) mode is the highest privilege and is responsible for interacting with hardware, which includes setting up and configuring the PMP registers. This is what privilege the *Security Monitor* executes on. There is no virtualization on this privilege, thus all memory accesses in this mode use physical addresses (assuming the default configuration).

Supervisor (S) mode is the privilege the host OS typically executes on. Keystone's *runtime* runs on supervisor privilege.

User (U) mode is what regular applications without any privilege execute on. All enclave applications run in this mode.

2.1.2 Keystone Components.

Security Monitor (SM): This is a small, trusted component, responsible for managing all the enclaves and for creating verifiable reports to prove that an application is running inside an enclave.

Runtime (RT): This component runs in S-mode and provides an interface for communicating to the SM from the user application. The runtime resides inside the enclave and is responsible for setting up the enclave application's environment (i.e. page table initialization).

Enclave Application (EAPP): This is the client's application that runs inside the enclave. For any resources the client requires from the host, it must interact with the runtime, which calls the SM.

Host Operating System: This is the OS of the host that uses the Keystone driver to interact with the SM to create or interact with the enclaves. This component is untrusted.

2.1.3 Virtual Memory Reliance.

In Keystone, page tables are managed and configured by the **runtime**, which is part of the enclave. We can completely remove the page table mapping altogether and perform no address translation. In this case, there is no reason to still have a **runtime**. This does not affect the memory isolation of the TEE because the isolation is provided by the PMP registers, which does not assume anything about the virtual memory of an enclave. This makes modifying Keystone to not use virtual memory a far easier task than SGX or TrustZone as Keystone's architecture does not rely heavily on virtual memory support.

2.2 FreeRTOS

Now that we have an appropriate enclave back-end to modify in order to run on an embedded device, we still require Real-Time Operating System (RTOS) functions to handle task scheduling.

FreeRTOS is an open-source real time operating system owned by Amazon and is a popular choice for embedded device developers. Amazon provides libraries to assist developers in connecting their device to use Amazon Web Services [20]. FreeRTOS has a very light memory footprint; in fact, there are only three source files in the kernel [15]. Furthermore, it provides user applications with useful data abstractions like queues for message passing between tasks

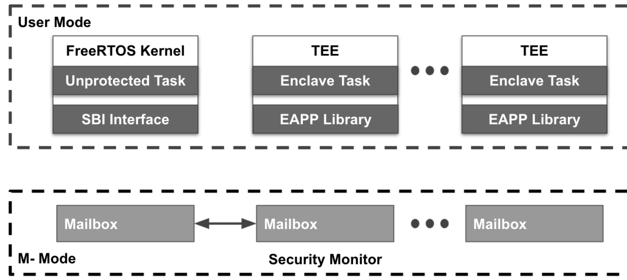

Figure 2. Design Overview of ERTOS

and synchronization primitives like a semaphore. FreeRTOS also provides several libraries for creating a TCP/IP stack or for IO support.

3 ERTOS Architecture

3.1 Design Overview

In order to create a framework for creating enclaves on embedded devices, we require a scheduler that can meet real time requirements. Keystone was not designed to be a scheduler. Its sole purpose is to create, delete, and switch into an enclave context. One can borrow several components of an RTOS kernel and port it to the SM, but that defeats the original point of the SM as being a light and trusted component that can be easily verified. We wanted to keep the SM lightweight as it was originally intended, while also providing the user with an RTOS kernel to schedule their tasks.

Because we removed virtual memory support, we only have Machine and User mode privilege. This would mean the FreeRTOS kernel, which was originally designed to run in S-mode, must run in user mode. This guarantees that even though the FreeRTOS kernel might be compromised, any tasks that are protected by an enclave will continue to be protected. This will virtually stop all privileged escalation attacks for compromising our framework. Furthermore, we minimize our total TCB by not having to trust the RTOS kernel.

We now present Enclaves in RTOS (ERTOS). ERTOS is a module for FreeRTOS that provides an API for creating and running enclaves dynamically, using Keystone's SM as a backend. A visual overview of our architecture can be seen in Figure 2.

We place the FreeRTOS kernel inside an enclave so that an embedded device can be remotely attested by the manufacturer or owner of the device to ensure it is using a correct and unmodified FreeRTOS kernel. We also have two types of tasks, *enclave* or *unprotected* tasks. Unprotected tasks do not have any hardware protection from other unprotected tasks and exist inside the FreeRTOS enclave. Enclave tasks live outside the FreeRTOS enclave and are protected by the SM. Both *enclave* and *unprotected* tasks will be scheduled by the FreeRTOS kernel. We support inter-task enclave communication as each enclave has a 512 byte mailbox that is maintained by the SM. Because message passing is handled by the SM, the SM can verify which enclave is sending a message and ensures the receiving enclave that a message it receives from its mailbox is authentic.

3.2 Bootloading

The only hardware modification Keystone requires is an embedded secret key, H_{sk}. We use the *Berkeley Bootloader*, which is a second-stage bootloader that originally booted Linux, but was modified to boot FreeRTOS. Upon CPU boot or reset, the bootloader will first initialize and generate a signed measurement of the SM with H_{sk}. It will then lock the SM region with a PMP register. The SM will in turn initialize and sign the measurement of the FreeRTOS kernel, then allocate a PMP region for it. Later, anyone can verify that the SM and FreeRTOS kernel was initialized correctly by observing the signed measurement.

Once the FreeRTOS kernel is initialized, it is free to create and schedule tasks. The enclave that holds the RTOS kernel is a special enclave that we call the *enclave scheduler*. The enclave scheduler has special privileges over other allocated enclaves. It has the ability to trap into the SM to create enclaves, to switch into other enclaves, and to enable or disable interrupts. The SM keeps track of which enclave is currently running and can verify the origin of any SBI call.

3.3 Task Registration

In order for an enclave task to be scheduled, it must first be registered with the SM. Registering an enclave task to the SM consists of sending the SM a request to create an enclave. The request will include the enclave entry point and the base and size of the enclave application. We assume the enclave application is already flashed into memory. Using the base and size of the enclave application allows the SM to allocate a PMP register to lock the enclave application. The SM will assign the enclave a unique task ID, which will then be returned to the FreeRTOS kernel as a handle to the enclave task. The task id will be used to schedule the enclave task later and is also used for sending and receiving messages. Once the enclave task is created, the FreeRTOS kernel can only switch into the enclave and cannot observe any internal state of the task, such as the registers. The SM does not handle any scheduling specific information, such as priority values for each task or the task state. We defer all scheduling policies to the FreeRTOS kernel.

3.4 Scheduling

The FreeRTOS kernel is responsible for creating and scheduling enclaves. FreeRTOS uses a task list to keep track of all tasks that can be scheduled. When the kernel wants to switch to another task, it will choose the task with the highest priority from the task list. We modified the FreeRTOS

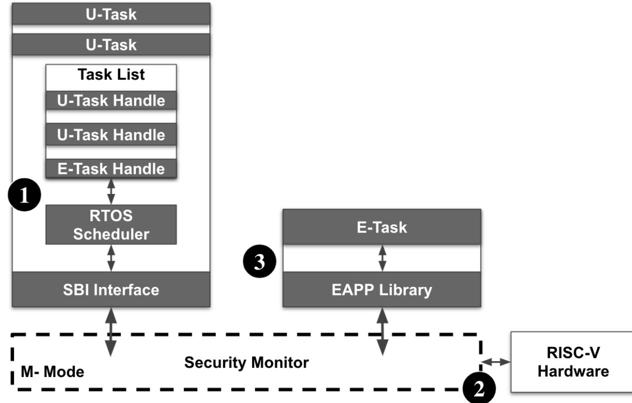

Figure 3. Enclave Scheduling Diagram. **E-Task** and **U-Task** refer to Enclave Task and Unprotected Task respectively.

kernel to also schedule tasks protected by an enclave. All scheduling policies will be completed from the FreeRTOS kernel. We illustrate the scheduling process in Figure 3.

Figure 3 ❶ The kernel selects a task and does an SBI call to trap into the SM. The kernel passes the task ID which the SM should switch into. Figure 3 ❷ The SM will then verify that the enclave scheduler sent the SBI request. Any calls to switch enclaves that does not originate from the enclave scheduler will cause the SM to switch back to the enclave scheduler. We chose to only allow the enclave scheduler to switch to another enclave to preserve the semantics of the FreeRTOS kernel. In a traditional RTOS, tasks must relinquish control to the kernel. In similar fashion, non-scheduling enclave tasks can only relinquish control back to the enclave scheduler. Figure 3 ❸ Upon an enclave task switch, the SM will switch the PMP registers to allow access to the memory region specified in the new enclave. The PMP registers that allow access to the SM and the enclave scheduler will be locked to guard against potential memory access by the newly switched in enclave.

We also allow enclave tasks to hint to the SM to allow directly switching to another enclave that it trusts. This is an optimization to avoid switching back to the FreeRTOS kernel if the enclave task expects frequent context switching or message passing to another enclave.

3.5 Interrupt Handling

FreeRTOS uses queues to service interrupts. Specifically for the FreeRTOS port in RISC-V, interrupts are registered via a vector table and the pointer to the vector table is stored in register `mtevec`. Upon an asynchronous interrupt, the `mcause` register is analyzed and used to decode how to handle the interrupt. The appropriate entry in the vector table is then chosen to branch to (given by `mtvec`). Since the FreeRTOS kernel is delegated to user mode, the SM handles interrupts and exceptions in similar fashion to the RTOS kernel. Currently, to mitigate DoS attacks, we setup a machine timer interrupt upon FreeRTOS kernel scheduler initialization. Any user-mode enclave must handle the timer interrupt and it cannot be ignored. If a user-mode enclave is running and uses up a lot of resources, the SM can step in and kill the enclave once the enclave's time quanta completes. Currently, the way the SM handles a machine-mode timer interrupt is to always switch into the enclave scheduler.

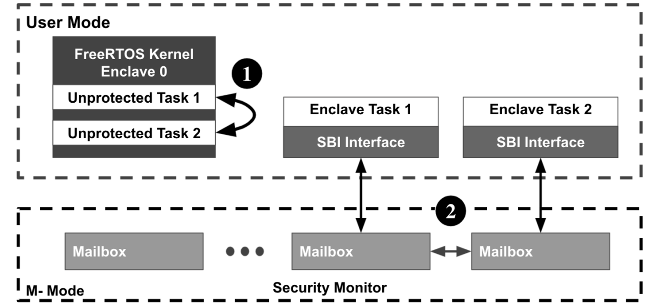

Figure 4. Message Passing Visualization

3.6 Message Passing

3.6.1 Small Message Passing.

FreeRTOS supports inter-task communication via queues. In order to support inter-enclave communication, we implemented per-enclave mailboxes. Each enclave has a statically allocated mailbox buffer that is managed by the SM. We must have the SM involved when sending these messages as the SM intervention ensures that if an enclave receives a message from enclave A, then enclave A actually sent the message. The SM guarantees that the message is sent from the purported enclave and the message contents are not seen from anyone other than the sender, recipient, and the SM. The mailbox interface was only meant for small (< 512 bytes) message passing between enclaves.

3.6.2 Bulk Message Passing.

For bigger messages, the SM can allocate an enclave buffer that is only accessible by the sender and receiver enclave. If there is no harm in leaving the shared buffer unprotected, the shared buffer can also be allocated outside of the enclave. Enclaves will be able to send messages directly without SM intervention.

3.7 Unprotected Tasks

Not all tasks will require strong isolation guarantees. For example, multiple tasks that communicate frequently with each other do not require strong isolation guarantees between them. Message passing between enclaves require SM intervention, which requires trapping into machine mode, flipping the PMP registers, and switching in the new enclave registers. This causes unnecessary overhead if isolation is

not required. This is further visualized in Figure 4. For this reason, we also add the notion of *unprotected tasks*.

❶ Unprotected tasks run in the same enclave as the enclave scheduler, so switching or sending messages between unprotected tasks do not require communication with the SM.

❷ Enclave tasks context switching or sending messages between each other must signal to the SM. Message passing requires copying the message from the enclave's buffer to its mailbox stored in the SM.

Even though unprotected tasks run in the same TEE as the FreeRTOS kernel, unprotected tasks are still prevented from gaining any confidential information in an enclave. A unprotected task may overwrite or modify the initial state of an enclave (i.e. enclave entry point), but this would be detectable because an enclave's initial state (enclave entry point, start address, and size) is hashed as well as each page of the enclave. Furthermore, an unprotected task cannot access another enclave task's memory because of the strong isolation guaranteed by the PMP registers. At worst, the unprotected task may be able to cause the FreeRTOS kernel to crash and prevent any enclaves from running, but preventing DoS attacks is beyond the scope of our trust model.

4 Implementation

Our implementation added around 1000+ LOC to the FreeRTOS Kernel and over 500+ LOC to Keystone's Security Monitor. The modifications included porting FreeRTOS to run in user-mode, creating APIs to create and interact with other enclaves, and modifying the existing FreeRTOS API to integrate tasks and enclaves. We also ensured that our enclave APIs were were similar to the existing task APIs in FreeRTOS. This would allow better familiarity with our API for embedded software developers. ERTOS can be compiled for RISC-V 32-bit (RV32GC) or 64-bit (RV64GC) machines.

5 Evaluation

For all benchmarks, we simulate a single RV64GC core, out-of-order machine (Berkeley's Out of Order Machine [8]) with DDR3 using FireSim [12]. The specific configuration in FireSim used is

```
firesim-boom-singlecore-no-nic-l2-llc4mb-ddr3
```

5.1 Micro-benchmarks: Inter-Task Communication

Due to the strict memory isolation between enclaves, message passing requires copying the message twice per message. First, the enclave must copy the message to its mailbox, then the receiver must copy the message from the mailbox to its own memory space. Currently, we use **asynchronous** message passing, where the users must poll the mailbox if they are waiting on a message.

A study found that **inter-task** communication accounted for the most frequent type of task communication (over forward and backward intra-task communication) [13]. Because of the importance of inter-task communication, we measured the total cycles required to send a small message between two tasks, using different configurations. We define our message passing test as a task sending a message to another task, then waiting for a reply. We will measure the number of cycles it takes for a full round trip.

For the `baseline`, both tasks will run within the same enclave domain. We will compare it to the performance of both tasks in separate enclaves. We get an average roundtrip cost of *4324* cycles for the *baseline*. Because varying the size of the message doesn't matter for the *baseline*, we use this average to calculate the slowdown for enclave message passing.

5.2 Optimizations

5.2.1 Synchronous Message Passing.

To minimize message copying, we implemented synchronous message passing. The sender of a message only sends to an enclave who has previously requested a message already. This would only require a single message copy as the SM can copy the message directly to the receiver's specified buffer. Furthermore, the receiver no longer has to poll for a message. Upon calling `sbi_recv`, the SM will set a flag to indicate that the enclave is waiting for a message then immediately switches to the RTOS enclave. When the sender enclave sends the message to the receiver, the sender will copy the contents directly to the receiver's buffer, then the receiver will be able to return from its initial call to `sbi_recv` without having to poll the SM any further.

5.2.2 Shared Buffer.

We also introduce another form of message passing, which allocates a shared buffer between both enclaves. Both enclaves can directly read/write from the buffer. When enclave A wants to send a message to Enclave B, Enclave A will store the message in the shared buffer and signal to the SM to switch to Enclave B. If needed, the buffer can be protected by allocating another PMP region and securing the buffer in an enclave, but this will consume a PMP register. To get around the PMP register cost, the SM can grant a temporary PMP register to lock a memory space for the sender and receiver enclave.

5.2.3 Results.

The average overhead of asynchronous message passing is a a 2.80x slowdown. We lower our average overhead to 2.40x and 1.62x for the synchronous and shared buffer message passing respectively. We observe that both asynchronous and shared message passing approaches provide a fair improvement to asynchronous message passing. Message passing through the shared buffer had the best performance as the context switching time is minimized. This is due to the SM only having to context switch to the receiver enclave, whereas for synchronous message passing, the SM

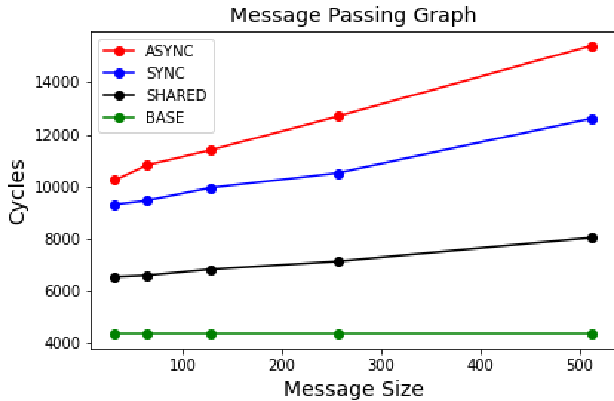

Figure 5. Message Passing Results Graph

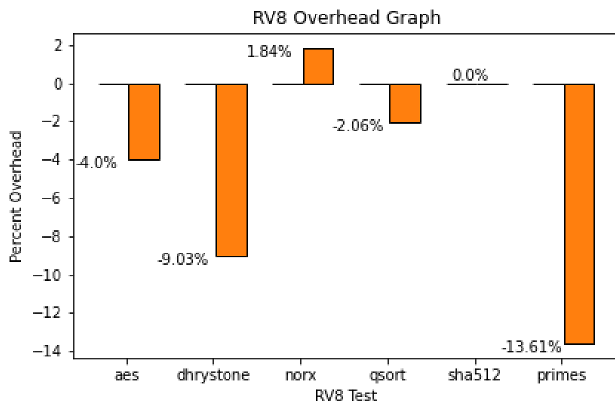

Figure 6. Graph that shows the percent overhead of ERTOS. Note: The **sha512** test isn't visible because the baseline and ERTOS took the same amount of time.

has to do additional processing (i.e. find the correct mailbox) and then copy the message to the receiver's mailbox.

5.3 RV8

To observe any computational overhead, we ported the RV8 benchmarks, which are compute-bound workloads, into our RTOS framework [9]. We measure the total time it takes to run the benchmarks inside an enclave versus running the benchmarks as an unprotected task in FreeRTOS. I exclude *miniz* and *bigint* for our evaluation because *miniz* had a large memory requirement and *bigint* relied on the C++ run-time, which our framework does not yet support.

Observing Figure 6, we found that the results using ERTOS had slightly better performance compared to the baseline. On average, ERTOS was faster by 5%. For enclaves, we pre-allocate a heap and each enclave runs our version of `malloc`, so that the enclave does not have to rely on FreeRTOS for `malloc` as this would incur a context switch to the FreeRTOS kernel. The malloc inside the enclave has a lower memory footprint and does not support coalescing heap blocks. The unprotected tasks use `malloc` provided by FreeRTOS which does support coalescing, which may cause memory de-allocation to be slower. This was the main attribute to the slight difference in performance. Furthermore, the heap for the baseline is shared with the FreeRTOS kernel, whereas the enclave application has its own private heap. This could make it more likely that the baseline could have poor cache locality as the heap might be fragmented due to sharing with the FreeRTOS kernel. We observe that there are no significant compute slowdowns when running the RV8 benchmarks on ERTOS.

6 Future Work

In our vision, we wanted to create a framework that can easily support enclave creation on almost all embedded devices that support TEE creation like SGX or TrustZone. Effectively, the vision for this project is to be able to provide a generalized API in FreeRTOS to create secure TEEs similar to OpenEnclave [2] or Asylo [1] that is agnostic of the hardware backend. Some of the challenges of this is that different hardware architectures provide somewhat different security guarantees. For example, creating a TEE in ARM TrustZone will not allow for strict isolation between enclaves [19]. We must be careful to annotate the different security guarantees, while also providing a generic API for different hardware back ends.

7 Conclusion

As we face a growing popularity of edge computing and IOT devices, there is an increased need for strong hardware isolation on embedded devices. We create ERTOS, a module in FreeRTOS that allows the creation of dynamic, secure tasks that can be attested and isolated from other tasks using Keystone as an enclave backend. Because of the strong isolation guarantees between enclaves, the overhead of message passing between enclaves is significant; however, we achieve negligible performance overhead when running compute intensive workloads. As we continue to develop ERTOS, we will actively investigate how to increase the performance of inter-enclave communication. In the future, we hope to provide a hardware agnostic module that will provide APIs to create TEEs across several enclave backends.

References

[1] [n.d.]. Asylo. https://asylo.dev/, note = Accessed: 2021-05-1.

[2] [n.d.]. OpenEnclave Switchless. https://openenclave.io/sdk/, note = Accessed: 2021-05-10.

[3] [n.d.]. Samsung SmartThings Developer Overview. https://docs.smartthings.com/en/latest/getting-started/overview.html, note = Accessed: 2021-05-11.

[4] [n.d.]. SmartApps Automation Community. https://community.smartthings.com/c/smartapps/created-smart-apps/43, note = Accessed: 2021-06-9.

[5] [n.d.]. TBONE – A zero-click exploit for Tesla MCUs. https://kunnamon.io/tbone/tbone-v1.0-redacted.pdf, note = Accessed: 2021-05-10.

[6] [n.d.]. TizenRT: Lightweight RTOS-based platform to support low-end IoT devices. https://github.com/Samsung/TizenRT, note = Accessed: 2021-06-9.

[7] Gui Andrade, Dayeol Lee, David Kohlbrenner, K. Asanović, and D. Song. 2020. Software-Based Off-Chip Memory Protection for RISC-V Trusted Execution Environments.

[8] K. Asanović, D. Patterson, and Christopher Celio. 2015. The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor.

[9] M. Clark. 2017. rv 8 : a high performance RISC-V to x 86 binary translator.

[10] Victor Costan and S. Devadas. 2016. Intel SGX Explained. *IACR Cryptol. ePrint Arch.* 2016 (2016), 86.

[11] Xingbin Jiang, Michele Lora, and Sudipta Chattopadhyay. 2020. An Experimental Analysis of Security Vulnerabilities in Industrial IoT Devices. *ACM Trans. Internet Technol.* 20, 2, Article 16 (May 2020), 24 pages. https://doi.org/10.1145/3379542

[12] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanovic. 2018. FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 29–42. https://doi.org/10.1109/ISCA.2018.00014

[13] Simon Kramer, Dirk Ziegenbein, and Arne Hamann. 2015. Real world automotive benchmarks for free. In *6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*.

[14] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanovic, and Dawn Song. 2020. Keystone: An Open Framework for Architecting Trusted Execution Environments. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*. https://arxiv.org/pdf/1907.10119.pdf.

[15] Real Time Engineers ltd. [n.d.]. The FreeRTOS™ Reference Manual. https://www.freertos.org/implementation/main.html.

[16] Knud Lasse Lueth. 2018. State of the IoT 2018: Number of IoT devices now at 7B – Market accelerating. https://iot-analytics.com/state-of-the-iot-update-q1-q2-2018-number-of-iot-devices-now-7b/.

[17] G. Mullen and L. Meany. 2019. Assessment of Buffer Overflow Based Attacks On an IoT Operating System. In *2019 Global IoT Summit (GIoTS)*. 1–6. https://doi.org/10.1109/GIOTS.2019.8766434

[18] Sandro Pinto and José Martins. 2019. The industry-first secure IoT stack for RISC-V: a research project.

[19] Sandro Pinto and Nuno Santos. 2019. Demystifying Arm TrustZone: A Comprehensive Survey. *ACM Comput. Surv.* 51, 6, Article 130 (Jan. 2019), 36 pages. https://doi.org/10.1145/3291047

[20] Amazon Web Services. [n.d.]. FreeRTOS: Real-time operating system for microcontrollers. https://aws.amazon.com/freertos/.

[21] Andrew Waterman, Yunsup Lee, Rimas Avizienis, David A. Patterson, and Krste Asanović. 2016. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.9.* Technical Report UCB/EECS-2016-129. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-129.html

[22] wolfSSL. [n.d.]. https://www.wolfssl.com/.

[23] Wei Zhou, Le Guan, Peng Liu, and Yuqing Zhang. 2019. Good Motive but Bad Design: Why ARM MPU Has Become an Outcast in Embedded Systems.